VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

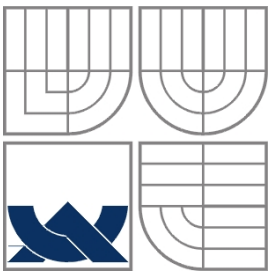
HOUGHOVA TRANSFORMACE PRO DETEKCI ČAR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

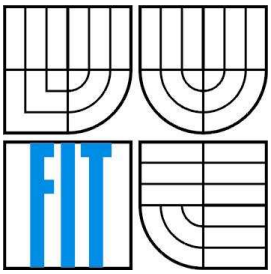
AUTOR PRÁCE
AUTHOR

Bořek Leikep

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

HOUGHOVA TRANSFORMACE PRO DETEKCI ČAR

HOUGH TRANSFORM FOR LINE DETECTION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Bořek Leikep
AUTHOR

VEDOUCÍ PRÁCE — Ing. Michal Španěl
SUPERVISOR

BRNO 2009

Abstrakt

Tato práce se zabývá využitím Houghovi transformace pro detekci čar. Teoreticky se dotýká také detekce kružnic a složitějších útvarů. Cílem práce je zhodnotit implementaci detekce čar v jazyce C# na platformě Microsoft .NET z hlediska především časové náročnosti. Práce obsahuje návrh a popis implementace programu Detektor, který je její součástí.

Abstract

This paper is describing use of the Hough Transform for line detection. It also contains theoretical description of circle detection and detection of more complex subjects. The goal of this work is to evaluate implementation of line detection in C# on Microsoft .NET platform mostly from a time consumption point of view. The paper contains description of implementation of the application Detektor which is part of this work.

Klíčová slova

detekce čar, detekce kružnic, Houghova transformace, C#, akumulátor, souřadný systém, detektor hran, úhel theta, práh intenzity, relativní intenzita, polární souřadnice přímky, časová náročnost

Keywords

line detection, circle detection, Hough Transform, C#, accumulator, coordinate system, edge detector, theta angle, intensity threshold, relative intensity, polar line coordinates, time consumption

Citace

Leikep Bořek: Houghova transformace pro detekci čar,
bakalářská práce, Brno, FIT VUT v Brně, 2009

Houghova transformace pro detekci čar

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Michala Španěla. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Bořek Leikep
18.5.2009

Poděkování

Chtěl bych tímto poděkovat svému vedoucímu Ing. Michalu Španělovi za odbornou radu a pochopení při zpracování této práce. Také bych chtěl poděkovat Ing. Jiřímu Venerovi za teoretické uvedení do problematiky detekce pravidelných křivek.

Obsah

1 Úvod

V dnešním světě informačních technologií zaujímá počítačová grafika význačné postavení. Počítačová grafika je hlavním prostředkem komunikace mezi softwarem a uživatelem. Zde se jedná v drtivé většině případů o výstup softwaru, který je zobrazen na výstupním zařízení tak, aby jej uživatel byl schopen snadno interpretovat. Nejedná se však o jediný způsob, jak využít počítačovou grafiku. Grafické rastry, případně vektory, mohou sloužit také jako vstupní data od uživatele do softwaru. V této oblasti se nabízí celá řada využití, od rozpoznávání lidského obličeje na vašem fotografickém aparátu, přes automatický převod psaného písma do textového editoru, až po např. rozpoznávání státních poznávacích značek na vozidlech. Jedná se o velmi složitou a rychle se rozvíjející oblast počítačové grafiky. Tato práce se zabývá jedním ze základních nástrojů v této problematice.

Houghova transformace je technika, chcete-li technologie, která umožňuje nalezení matematicky definovaných struktur v rastrovém obraze. Tento text se z valné části zabývá využitím této techniky pro detekci rovných čar. Lehce se dotýká obecné Houghovi transformace, která v počítačové grafice nalézá uplatnění, je však podstatně komplexnější na implementaci. Nejčastěji se Houghova transformace využívá pro detekci čar, kružnic, elips a dalších lehce matematicky definovatelných útvarů. Pro tuto oblast se často používá označení „klasická Houghova transformace“. Implementace v tomto případě není tak obtížná, ačkoli může být náročná na zdroje hardwaru.

Klasická Houghova transformace může mít nejrůznější druhy implementace, určené pro nejrůznější hardware. I mezi implementacemi určenými pro stejný typ hardwaru (např. osobní počítač) může být značný rozdíl, v závislosti na použitém programovacím jazyce. Tato práce si klade za svůj cíl zhodnocení možnosti využití technologie Microsoft .NET a jazyka C# jako implementačního prostředku pro Houghovu transformaci a porovnání dosažených výsledků s jinou implementací. Je logické předpokládat, že implementace pomocí jazyka C#, který je zcela objektový, bude především časově náročnější. Režie objektového kódu na Von Neumannově architektuře je vskutku prokazatelně náročnější, než je tomu u klasického procedurálního kódu. Otázka zní jak moc a jak moc se tento rozdíl projeví při implementaci složitější grafické operace, jakou je Houghova transformace.

Za tímto účelem vznikl program Detektor, který je součástí této práce. Při implementaci tohoto programu bylo využito volně dostupných knihoven AForge.NET (http://code.google.com/p/aforge/), které jsou určeny pro použití v prostředí Microsoft .NET. Tyto knihovny krom jiného také obsahují implementaci Houghovi transformace pro detekci čar a program Detektor umožňuje její srovnání s implementací napsanou autorem této práce v jazyce C#. Je třeba mít na paměti, že se jedná o srovnání velmi sofistikovaného a dobře odladěného nástroje s implementací, která vznikla během několika málo týdnů. Tak je také třeba nahlížet na dosažené výsledky při hodnocení a vyvozování závěrů. Program Detektor umožňuje přímé srovnání časové náročnosti detekce přímých čar a subjektivní porovnání výsledných dat. Subjektivní proto, že struktura výsledných dat je u obou implementací značně odlišná a vhodnost jejich využití silně závisí na vnějších okolnostech.

2 Houghova transformace

Jak již bylo zmíněno v úvodu, Houghova transformace je technika, sloužící k nalezení struktur v obraze. Jedná se o metodu, která nachází široké uplatnění v počítačové grafice. Vyžaduje, aby struktury, které chceme hledat, byly definovány v parametrické formě. Z tohoto důvodu se využívá především pro detekci útvarů, jako jsou přímky, kružnice, elipsy apod., u kterých je parametrický popis dobře známý a snadno formulovatelný. Tuto oblast označujeme termínem „klasická Houghova transformace“, která nalézá využití např. v medicínské grafice, kde hranice mezi objekty jsou často tvořeny těmito jednoduchými útvary. Výhodou Houghovi transformace je fakt, že velmi spolehlivě funguje i s nekvalitními vstupními daty, především pokud se jedná o šum.

Vstupem bývá obraz již zpracovaný detektorem hran a jinými morfologiemi. Tento postup je výhodný, protože v drtivé většině případů zvyšuje pravděpodobnost nalezení hledaných struktur a významným způsobem snižuje časovou náročnost celé detekce. Ovšem i z dokonale připravených vstupních dat není jednoduché jednoznačně označit hledané útvary. Cílem Houghovi transformace proto je provést zhodnocení situace a navrhnout kandidáty, ze kterých je poté možno vybírat.

Výstupem je soubor struktur v parametrické formě. Tyto struktury můžeme nazývat kandidáty. Tito kandidáti obvykle nesou dodatečnou informaci o tom, jak si stojí vůči ostatním kandidátům. Nejčastěji hovoříme o relativní intenzitě vzhledem k průměru, mediánu či maximu.

Omezením Houghovi transformace je fakt, že nedokáže nalézt útvary, které nejsou velmi intenzivní ve vstupních datech. Budeme-li mít např. Obrázek 1, ve kterém je celá řada nepravidelných, ale intenzivních útvarů a my budeme hledat obdélník, ve kterém je umístěno logo, pravděpodobně jej nenalezneme. Důvod je ten, že buď se vůbec nedostane do vstupních dat (jak ukazuje Obrázek 2) a detektor hran či jiný filtr jej zcela odstraní nebo tento útvar zanikne v „šumu“ ostatních intenzivnějších objektu, které budou částečně splňovat parametry hledaného útvaru. U útvarů, v jejichž parametrickém vyjádření figuruje vetší množství parametrů, bývá problém splynutí s okolím znatelnější, proto je třeba být opatrný, pokud se snažíme nalézt složitější útvary než přímky a kružnice.

Obrázek 1. Viditelný obdélník.

Obrázek 2. Po detekci hran obdélník zmizel.

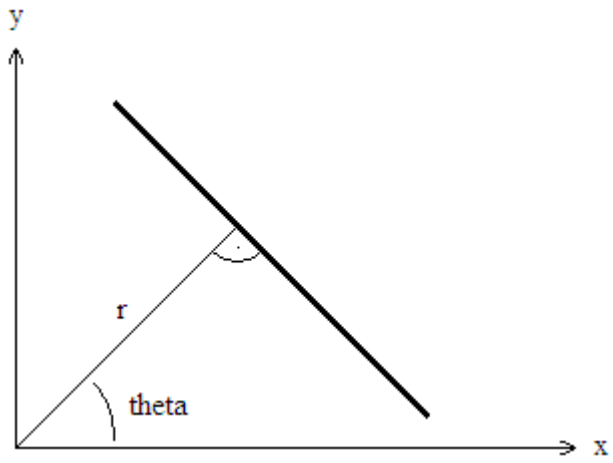
Transformace byla poprvé patentována v roce 1962 Paulem Houghem. Houghovu transformaci, tak jak se dnes využívá v počítačové grafice, zformulovali Richard Duda a Peter Hart v roce 1972 a nazvali ji „generalizovaná Houghova tranformace“ (generalised Hough Tranform), která nabrala na popularitě hlavně během osmdesátých let dvacátého století.

2.1 Detekce čar

Mluvíme-li o čáře, máme na mysli přímku. Jak již bylo zmíněno, potřebujeme přímku nejprve vyjádřit parametricky. Jednou z možností jak vyjádřit přímku v kartézské soustavě je vztah $y = a \cdot x + b$. Nevýhodou tohoto vyjádření je, že pro přímky rovnoběžné s osou Y se a stává nekonečným, proto je toto vyjádření nevhodné pro použití v Houghově transformaci. Daleko častěji se používá tzv. polární tvar parametrického vyjádření přímky

$$r = x \cdot cos\theta + y \cdot sin\theta \qquad y = \left(-\frac{cos\theta}{sin\theta}\right)x + \left(\frac{r}{sin\theta}\right) \qquad \textbf{Rovnice 1 a 2.}$$

kde r je vzdálenost přímky po kolmici od počátku a θ (theta) je úhel, který tato kolmice svírá s kladnou osou X. Pro účely Houghovi transformace úhel θ nabývá hodnot od 0 do π radiánů. Situaci lépe znázorňuje Obrázek 3.



Obrázek 3. Polární definice přímky.

Při detekci se využívá tzv. *akumulátor*. Pro detekci čar je akumulátor dvojrozměrný graf, kde je na jedné ose vynesena vzdálenost r a na druhé ose úhel θ. Každý bod grafu navíc také nese dodatečnou informaci o své „intenzitě“. Při implementaci se jako akumulátor používá dvojrozměrné pole celočíselných proměnných.

Postup detekce je takový, že pro každý bod ve vstupních datech se provede cyklus, při kterém se ukládají hodnoty r a θ do akumulátoru. V tomto cyklu se do výše zmíněné rovnice doplní hodnoty x a y, které korespondují se souřadnicemi aktuálního bodu v bitmapě a úhel θ v rozpětí (proto cyklus) od 0 do π po předem dohodnutých krocích. Dostaneme vzdálenost od počátku r. Do akumulátoru na pozici odpovídající hodnotě r a všech hodnot úhlu θ přičteme dohodnutou hodnotu (nejčastěji 1) a tím zvýšíme jeho intenzitu. Tento cyklus se provede pro každý bod ve vstupních datech. Na konci budeme mít naplněný akumulátor. Obrátek 4 ukazuje příklad grafického znázornění naplněného akumulátoru.

Obrázek 4. Příklad naplněného akumulátoru.

Zde je možno vidět, že akumulátor obsahuje maxima (na Obrázku 4 nejsvětlejší body). Tyto maxima vznikla tím, že ve více cyklech (jeden cyklus na jeden vstupní bod) došlo k uložení popisu (hodnoty r a θ) stejné přímky. Pokud několik vstupních bodů leží na přímce, každý z nich uloží do akumulátoru popis (mimo jiné) právě oné přímky, na které leží ostatní body. Tím vznikne maximum.

Z těchto vzniklých maxim můžeme lehce zformulovat kandidáty na nalezené čáry. Z maxima odečteme hodnoty r a θ, které dosadíme do rovnice $y = \left(-\frac{cos\theta}{sin\theta}\right)x + \left(\frac{r}{sin\theta}\right)$, čímž dostaneme klasický předpis pro přímku $y = a \cdot x + b$. K těmto kandidátům můžeme připojit informaci o jejich intenzitě, chcete-li pravděpodobnosti skutečného výskytu. Tady máme více možností, jak se k této hodnotě dostat. Nejčastěji se však používá tzv. relativní intenzita vůči průměru, mediánu nebo maximu. Budeme-li pro příklad uvažovat poslední z možností, hodnotu spočteme jednoduše tak, že podělíme intenzitu našeho maxima intenzitou nejvýraznějšího maxima v akumulátoru. Dostaneme tak hodnotu někde mezi 0 a 1. Při zpracovávání kandidátů pak můžeme jednoduše např. vznést podmínku, že nás zajímají pouze kandidáti s relativní intenzitou 0.8 a vyšší. Tím dostaneme pouze ty „pravděpodobnější“ kandidáty a omezíme jejich celkové množství.

Časovou náročnost detekce lze snížit aplikováním vhodných filtrů na vstupní obrázek. Nejčastěji se používá detektor hran (např. Canny operator). V případě přímek je možné aplikovat další filtry jako třeba prahování, erozi a jiné. Je třeba mít ale na paměti, že může dojít ke ztrátě kritických dat a tím k neúplnému nebo zkreslenému výsledku.

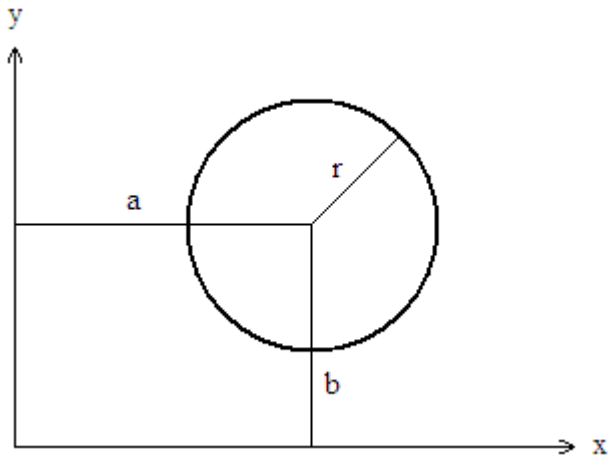
Klíčovým parametrem u detekce přímek je krok úhlu θ, který má zásadní vliv na časovou a paměťovou náročnost detekce a také na kvalitu výsledku. Pokud bude tento krok příliš malý, je možné, že dojde k „přehlédnutí“ některé čáry proto, že jednoduše leží pod takovým úhlem, do kterého se cyklus detekce nedostane. Např. náš krok bude 45˚, tak cyklus detekce bude zkoumat pouze přímky ležící pod úhly 0˚, 45˚, 90˚ a 135˚. Pokud ale bude v obrázku čára pod úhlem 20˚, detekce ji do akumulátoru nevloží nebo vloží jen její část, která ale bude ležet pod špatným úhlem. Proto je velmi důležité zvolit správnou velikost kroku úhlu θ.

2.2 Detekce kružnic

Při detekci kružnic postupujeme obdobně jako u detekce čar, je zde ale rozdíl v komplexnosti celé operace. Podívejme se ale nejprve na parametrické vyjádření kružnice, které nám napoví více.

$$(x-a)^2 + (y-b)^2 = r^2 \qquad \textbf{Rovnice 3.}$$

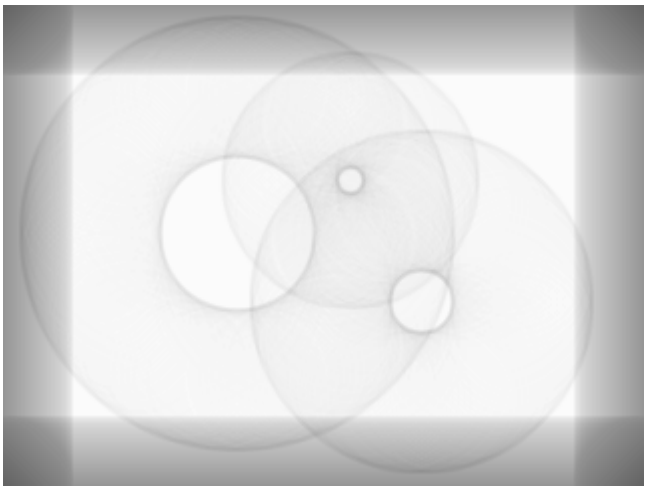
kde a a b jsou souřadnice středu kružnice a r je poloměr. Obrázek 5 znázorňuje situaci.



Obrázek 5. Definice kružnice.

Jak tedy můžeme vidět, na rozdíl od přímek, máme u kružnic tři parametry v parametrické rovnici. Z toho vyplívá, že akumulátor, který budeme potřebovat na Houghovu transformaci, bude trojrozměrný. To má obrovský vliv na časovou a paměťovou náročnost detekce. Parametr r značně komplikuje celou situaci. Do obrázku mohou zasahovat opravdu velké kružnice, které mají poloměr několikanásobně větší, než jsou jeho rozměry. Nejen že by rozměr r v akumulátoru musel být dostatečně velký pro takovéto kružnice, rozměry a a b by musely mýt neméně rozměrné, protože středy takovýchto obřích kružnic by byli daleko za hranicemi původního obrázku. Takto velký trojrozměrný akumulátor by poté zcela znehodnocoval výhody Houghovi transformace. Z tohoto důvodu se omezuje maximální (i minimální) poloměr kružnic, čímž se omezí velikost všech tří rozměrů akumulátoru a tím drasticky sníží časové a paměťové nároky detekce.

Cyklus detekce je v případě kružnic poněkud odlišný od přímek. Kolem každého vstupního bodu kreslíme pomyslné kružnice o různých poloměrech. Souřadnice vstupního bodu se dosadí jako proměnné a a b do parametrické rovnice. Poté v cyklu měníme poloměr r po předem dohodnutých intervalech a postupně počítáme hodnoty x a y, které splňují rovnost v rovnici. Na tyto souřadnice do aktuální hloubky r v akumulátoru přičítáme (zvyšujeme intenzitu). Příklad naplněného akumulátoru můžeme vidět na Obrázku 6. Nezapomeňte, že jde jen o jednu rovinu v trojrozměrném akumulátoru.



Obrázek 6. Akumulátor při detekci kružnic.

Pro lepší pochopení si představte akumulátor jako kvádr, ve kterém jsou útvary, které by vznikly spojením dvou jehlanů v jejich vrcholech. Co vidíme na Obrázku 6 je to, co uvidíme, kdybychom rozřízly tento kvádr v náhodném místě. V místě, kde se naše jehlany setkávají, vzniká maximum. Toto maximum nám poskytuje tři informace (tři rozměry), které můžeme dosadit do rovnice kružnice. Jedná se o proměnné a, b a r. Řešením rovnice jsou pak takové hodnoty x a y, které představují danou kružnici. Tyto maxima opět tvoří kandidáty, kteří mohou nést dodatečnou informaci o intenzitě maxima, chcete-li o „pravděpodobnosti“ kandidáta.

Velikost akumulátoru by měla být taková, aby dokázala postihnout i kružnice, jejichž střed leží mimo obrázek. Zde přichází ke slovu maximální poloměr kružnice, kterou chceme hledat. Stanovuje tak maximální velikost dvou rozměrů akumulátoru, protože nemá smysl hledat kružnice, které nezasahují do obrázku, protože takové zcela jistě neexistují. Velikost třetího rozměru je závislá na maximálním a minimálním uvažovaném poloměru kružnice a především na kroku, po kterém se bude v cyklu detekce tento poloměr měnit. Menší krok znamená lepší výsledky, ale vetší akumulátor, tedy vyšší paměťovou i časovou náročnost detekce.

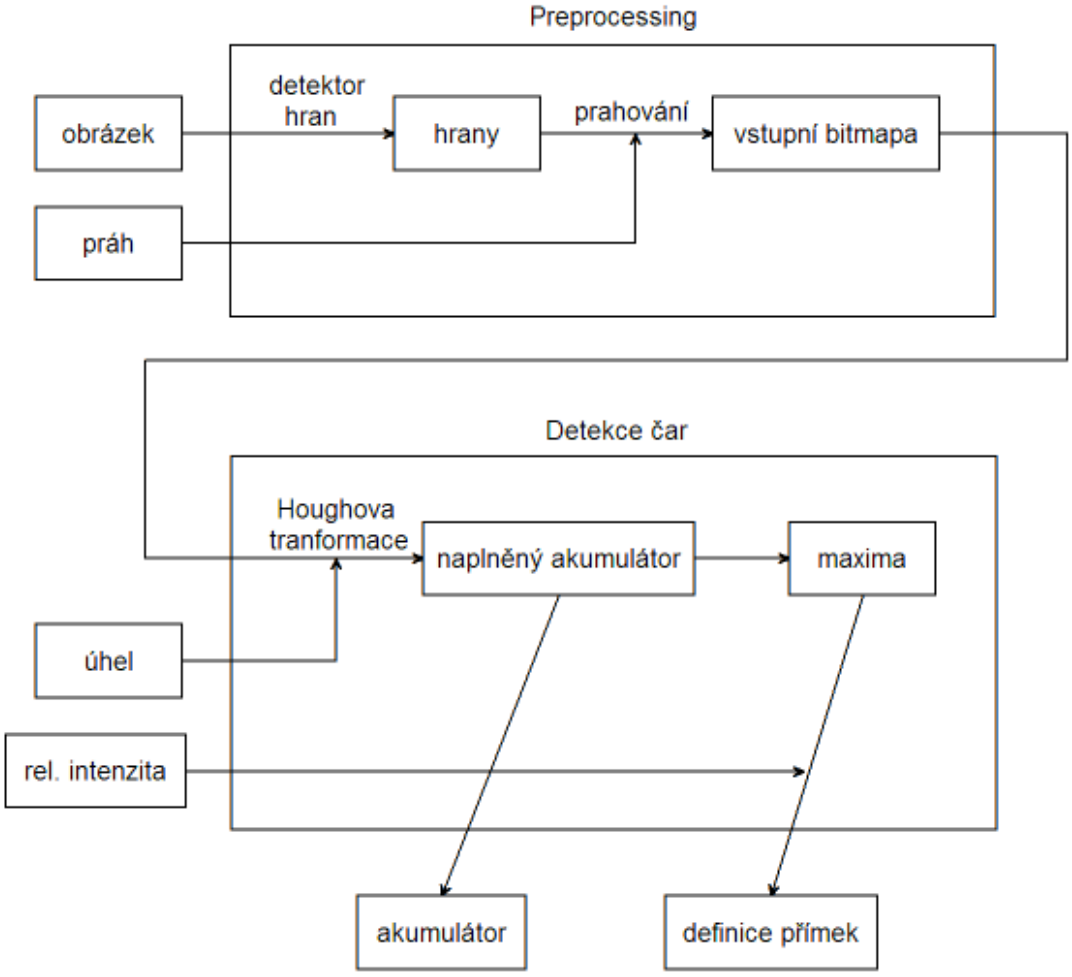
2.3 Detekce složitějších útvarů

Nyní se dostáváme do oblasti generalizované (obecné) Houghovi transformace. Budeme-li chtít detekovat složitější útvary, asi nebudeme analyticky schopni nalézt jejich parametrické vyjádření. V takovém případě se situace zjednodušuje tím, že se používá tabulka, která zachycuje jisté závislosti v hledaném útvaru. Tyto závislosti však nejsou definicí. Robustnost takovýchto charakteristik má zásadní vliv na to, zda nám detekce bude schopna nalézt požadované tvary.

Takováto tabulka s charakteristikami se vytvoří ještě před samotnou detekcí. Za tím to účelem je třeba dodat prototyp tvaru, který chceme hledat. Z tohoto prototypu se tedy vygeneruje tabulka charakteristik, která bude dále v detekci figurovat v podstatě stejně jako definice útvaru. Tvar akumulátoru je v takovýchto případech závislý na konkrétních matematických tvarech charakteristik, které jsou uloženy do tabulky.

3 Návrh implementace

Detekci čar si rozdělíme do dvou logických částí. Nejprve se bude provádět tzv. preprocessing, kde připravíme ideální vstupní data pro Houghovu transformaci. Potom proběhne detekce, po které již bude možné převzít výstupy. Obrázek 7 ukazuje diagram, který vyznačuje tento process.



Obrázek 7. Schéma implementace detekce čar.

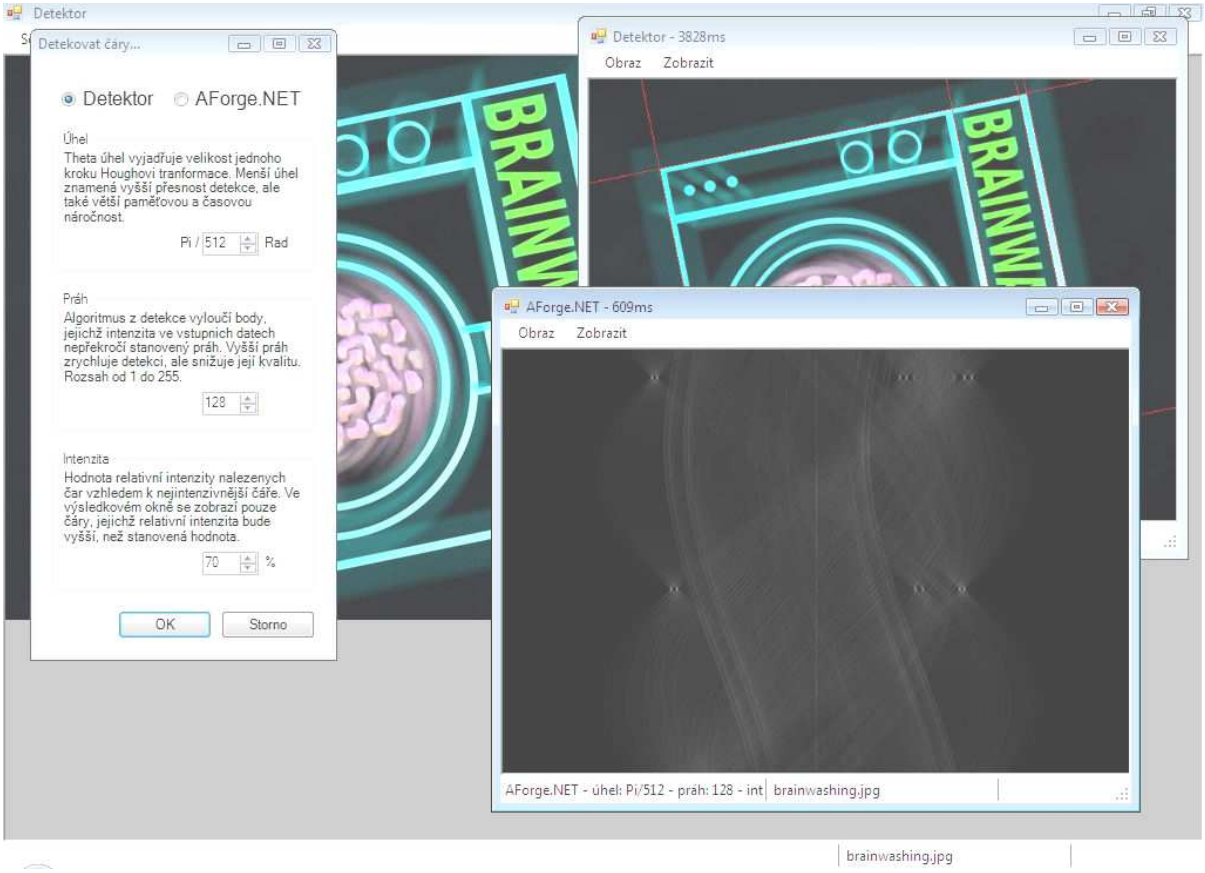
Prvním krokem je vložení vstupního obrázku. Následně proběhne detekce hran, jejímž výstupem bude černobílý obraz, znázorňující intenzitu jednotlivých hran. Nad tímto obrazem se provede prahování podle vstupního parametru. Po tomto budeme mít připravenou bitmapu pro Houghovu transformaci. Tímto končí preprocessing.

Následuje detekce čar, která sestává především v Houghově transformaci. Jejím vstupem je velikost kroku úhlu θ. Po transformaci budeme mít naplněný akumulátor a v této chvíli již můžeme sestavit grafickou podobu akumulátoru, což je jeden z výstupů. Následně musíme nalézt maxima v akumulátoru a vypočítat jejich relativní intenzitu. Nyní již můžeme sestavit definice přímek, které

chceme předat jako výstup. Klíčem v rozhodování bude vstupní parametr minimální relativní intenzity. Tímto končí celý algoritmus detekce čar a můžeme dále pracovat s jeho výstupy. Součástí implementace může být např. vykreslení výsledných přímek na výstupní zařízení nebo jejich další použití v jiných algoritmech.

4 Implementace detekce čar

Praktická část této práce spočívá v implementaci detekce čar. Za tímto účelem vznikl program Detektor, který umožňuje snadnou demonstraci použití této detekce. Je možno si vybrat, jaké knihovny se mají při detekci použít. Mohou to být buď knihovny, jejichž implementace je součástí této práce nebo se mohou použít knihovny AForge.NET. Dále program umožňuje nastavení základních parametrů detekce jako je velikost kroku úhlu θ a relativní intenzita výstupních kandidátů. Obrátek 7 ukazuje program Detektor v akci.



Obrázek 8. Program Detektor v akci.

Prostředí programu je napsáno v jazyce C#. Hlavní okno umožňuje otevřít soubor s obrázkem, na kterém budeme chtít provést detekci. Dává možnost přizpůsobit obrázek velikosti okna, zobrazuje nápovědu a stavový řádek s dodatečnými informacemi. Stisknutím tlačítka F5 nebo vybráním položky v menu zobrazíme dialogové okno detekce čar.

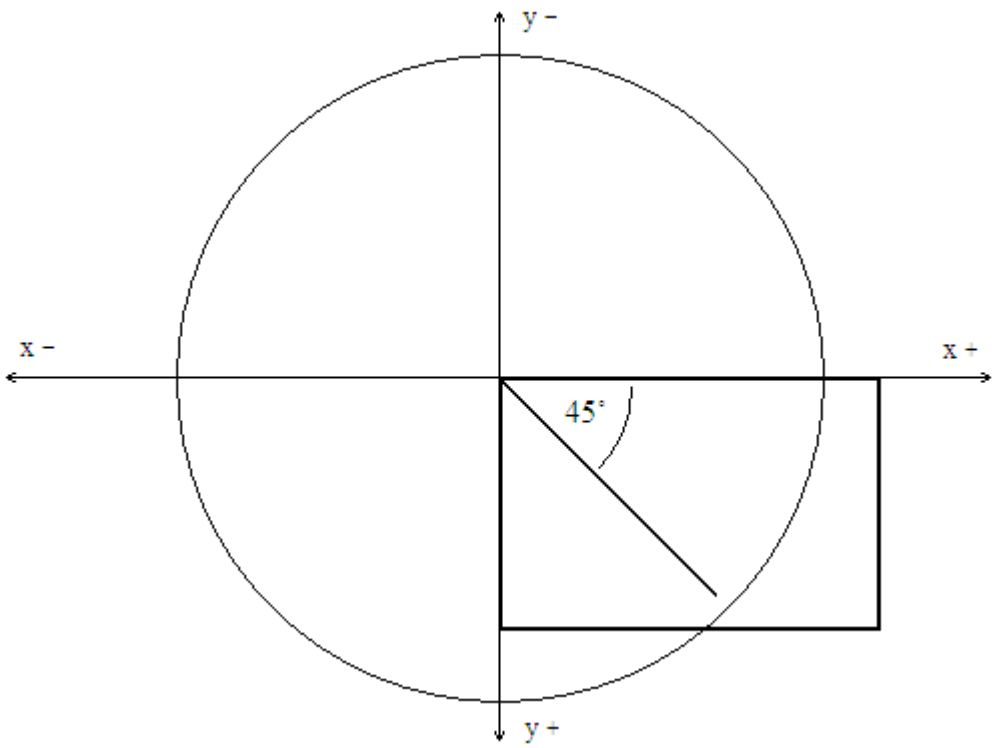
Dialogové okno detekce dává na výběr detekci Detektor (implementace autora práce) nebo AForge.NET. Dále umožňuje nastavit velikost kroku úhlu θ při detekci, respektive dělitel číslem π. Větší dělitel znamená menší úhel, což znamená lepší výsledky detekce, ale vyšší časová a paměťová náročnost. Maximální hodnota dělitele je 2048. Hodnota práh určuje, jakou intenzitu musí mít bod ve vstupních datech, aby byl do detekce zahrnut. Vyšší práh značně snižuje časovou náročnost operace, může však dojít ke ztrátě kritických dat. Intenzita určuje minimální relativní intenzitu výstupních kandidátů. Čáry, které budou splňovat tuto podmínku, se po detekci objeví ve výsledkovém okně. Uzavřeme-li dialogové okno tlačítkem OK, spustí se detekce a ve stavovém řádku se objeví upozornění. Během detekce může být problém s prací v hlavním okně.

Po skončení detekce se objeví výsledkové okno. Výsledkové okno implicitně zobrazí původní obrázek a nalezené čáry. V menu je však možno vybrat zobrazení původního obrázku, vstupních dat a akumulátoru. Také je zde možno vypnou či zapnout zobrazení výsledných čar. Vstupní data představují obrázek, který vstupuje do Houghovi transformace. Tento obrázek je již zpracován detektorem hran a filtrem prahování. Volba akumulátor zobrazí vnitřní akumulátor, ovšem Detektor a AForge.NET mají jinou odlišnou orientaci akumulátoru, proto se zdají být navzájem otočeny o 90˚. Orientace akumulátoru Detektoru je možno vidět na Obrázku 4 nebo na Obrázku 11. Výsledkové okno, tak jak okno hlavní, umožňuje přizpůsobit obrázek velikosti okna.

Ve stavovém řádku výsledkového okna nalezneme informace o vstupních parametrech detekce a o trvání detekce. Do délky trvání detekce se zahrnuje i fáze předzpracování vstupních dat pomocí detektoru hran a filtru prahování. Čas se zobrazuje v milisekundách.

4.1 Souřadný systém

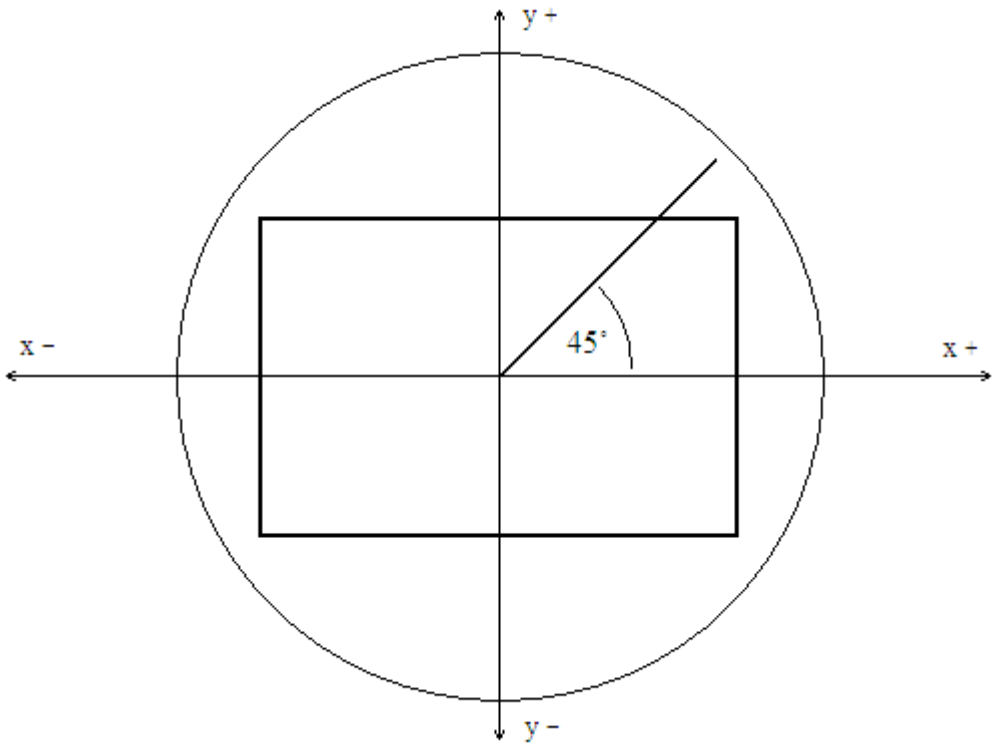
Zvolení správného souřadného systému hraje podstatnou roli při implementaci Houghovi transformace. Třídy programu Detektor používají souřadný systém podobný souřadnému systému, který se používá při vykreslování na obrazovku. Počátek je v levém horním rohu obrázku, kladná osa X jde od počátku doprava a kladná osa Y jde od počátku dolů. Právě proto, že osa Y směřuje dolů, úhel roste od osy X ve směru hodinových ručiček. Takovýto systém může být neobvyklý, perfektně však vyhovuje našim potřebám při detekci. Souřadný systém programu Detektor je ilustrován na Obrázku 9.



Obrázek 9. Souřadný systém programu Detektor.

Obrázek je pak pomyslně vložen do pravého dolního kvadrantu stejně tak, jak je zobrazený na obrazovce. V programu Detektor jsou úhly vždy uváděny v radiánech. Při vykreslování výsledných čar je třeba mít na mysli všechny tyto skutečnosti. Třídy programu Detektor poskytují kreslící funkci právě na tento souřadný systém.

Souřadný systém používaný knihovnami AForge.NET je odlišný. Kladná osa Y směřuje od počátku nahoru a úhel roste od kladné osy X proti směru hodinových ručiček. Čili klasický souřadný systém. Hlavní rozdíl mezi těmito dvěma souřadnými systémy je však v tom, že knihovny AForge.NET pokládají počátek souřadného systému do středu obrázku. Taktéž pracují s úhly ve stupních, nikoli v radiánech. Tento souřadný systém je ilustrován na Obrázku 10.



Obrázek 10. Souřadný systém knihoven AForge.NET.

Přesto, že oba systémy mají na výstupu stejný formát, jsou oba vztaženy k jinému souřadnému systému. Jelikož knihovny programu detektor obsahují funkci pro vykreslení přímek pomocí grafického kontextu, bylo mým cílem provézt konverzi výsledných přímek z knihoven AForge.NET.

Máme tedy dva souřadné systémy, ve kterých máme jeden obrázek. Postup je takový, že přiložíme tyto dva systémy (Obrázky 9 a 10) k sobě právě v místě, kde je umístěn obrázek. Z toho nám vznikne obrázek se čtyřmi souřadnými osami. Naším úkolem v tuto chvíli je převézt definici přímky z jednoho systému do druhého.

Nejprve je třeba převézt uhel θ ze stupňů do radiánů. To učiníme jednoduše tím, že stupně podělíme 180 a vynásobíme π. Dále si spočítáme přesné souřadnice jednoho bodu na přímce. Zvolíme-li $x = 0$, rovnice přímky se nám zjednoduší na $y = \frac{r}{sin\theta}$. Další krok je převedení tohoto bodu do cílového souřadného systému. Víme, že počátky souřadných systémů jsou od sebe vzdáleny o polovinu rozměrů obrázku, čili půlka šířky na ose X a půlka výšky na ose Y. Přičteme tedy k původním souřadnicím tyto hodnoty a navíc otočíme Y vynásobením -1. Nyní musíme dopočítat úhel θ. Jedná se v podstatě o doplněk do π, protože osy obou souřadných systému jsou rovnoběžné, pouze osa Y je převrácena. Takže $\theta_2 = \pi - \theta_1$. Posledním krokem v konverzi je dopočítání nové vzdálenosti od počátku r. Toho docílíme jednoduše tím, že dosadíme do rovnice $r = x \cdot cos\theta + y \cdot sin\theta$, kde x a y jsou souřadnice našeho bodu na přímce, ale již v cílovém souřadném systému.

4.2 Třída HLine a HLineCollection

Třída **HLine** reprezentuje přímku. Obsahuje informace nezbytné pro sestavení definice konkrétní přímky a obsahuje dodatečnou informaci o intenzitě. Při požadavku na vrácení detekovaných čar se definice přímek ukládají právě do této třídy. Tato třída rovněž obsahuje statickou metodu, která umožňuje vykreslení přímek do určeného prostoru a na konkrétní výstupní zařízení.

Proměnná *Theta* (typ double) nese informaci o úhlu, který svírá kolmice na přímku protínající počátek (neboli nejkratší spojnice) s kladnou osou X. Úhel je uložen v radiánech a jeho velikost se pohybuje mezi 0 a π, ovšem hodnoty π nikdy nenabývá, protože z pohledu definice přímky je úhel π totožný s úhlem 0. Jinými slovy přímka nemá orientaci. Jelikož vstupní parametr detekce krok úhlu θ nabývá pouze hodnot, které vzniknou dělením čísla π celočíselným dělitelem, budou proměnné Theta nabývat pouze těchto podílů.

Proměnná *Radius* (typ int) vyjadřuje délku spojnice přímky a počátku souřadného systému v pixelech. Tato spojnice je kolmá na přímku a je nejkratší možnou spojnicí. Tato vzdálenost normálně v analytické matematice není omezena na obor celých čísel, ovšem pro účely Houghovi transformace musí nabývat pouze celých čísel. Důvod je ten, že tato hodnota je jednou z dimenzí interního akumulátoru. Tento akumulátor je implementován jako paměťové pole, proto není možné pracovat s libovolnou desetinou hodnotou. Je však možné, aby jedna jednotka vzdálenosti byla v akumulátoru rozdělena na více částí a tím dosažení vyšší přesnosti. V programu Detektor tomu tak ale není, vzdálenost jednoho pixelu obrázku je reprezentována v akumulátoru pouze jednou hodnotou, a proto výsledná hodnota radius nabývá pouze celých čísel. Maximální hodnota a zároveň velikost jedné dimenze akumulátoru je délka úhlopříčky vstupního obrázku v pixelech. Přímky, které by byly po kolmé spojnici vzdálenější než toto maximum by nemohli zasáhnout do obrázku. Jedná se ovšem o maximální možnou absolutní vzdálenost přímky. To znamená, že vzdálenost od počátku může nabývat jak kladných tak záporných hodnot. Z tohoto důvodu musí být příslušná dimenze akumulátoru dvojnásobná, viz Obrázek 4 a Obrázek 11.

Proměnná *Intensity* (typ double) nese informaci o relativní intenzitě přímky. Jedná se o relativní intenzitu vzhledem k intenzitě nejvýraznější nalezené přímky. Hodnoty se pohybují od 0 do 1. Tato hodnota však není součástí definice přímky, jde pouze o dodatečnou informaci, kterou je možno využít dle potřeby.

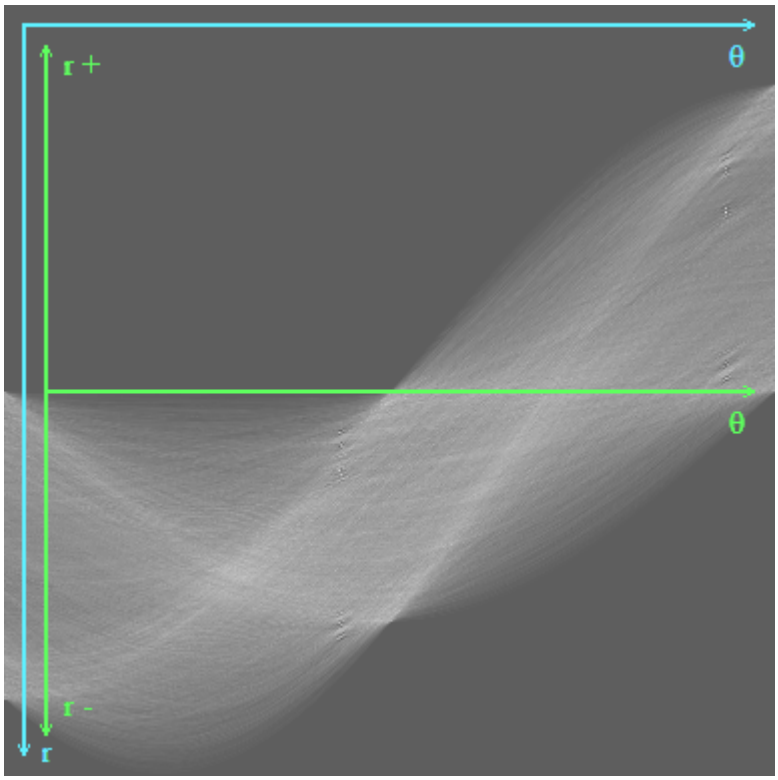
Metoda *DrawLines* umožňuje vykreslení přímek definovaných objekty HLine. Metoda je statická, proto není navázána na existenci objektů HLine ale přímo na třídu a lze ji proto využít kdykoli bez vytváření jakýchkoli objektů. Vstupem této metody je objekt třídy Graphics, který poskytuje grafický kontext (nejen) zařízení, do kterého se bude kreslit. Dalším vstupem je objekt třídy Pen, který určuje parametry kreslícího pera. Proměnná bounds definuje obdélník, který vyznačuje oblast, do které se bude kreslit (nejčastěji hranice obrázku). Posledním parametrem je seznam přímek k vykreslení.

Třída **HLineCollection** je kontejnerem pro objekty třídy HLine. Dědí od standardní třídy CollectionaBase a definuje základní metody pro vkládání, odebírání a vyhledávání v seznamu. Tato metoda seznamu je použita především proto, že se jedná o standardní způsob v prostředí .NET a jazyku C#. Alternativou je použít klasické pole, podobně jako třeba v jazyce C++. Cílem této práce je však porovnání časové náročnosti detekce napsané standardními nástroji prostředí .NET a detekce napsané pomocí některého procedurálního jazyka jako např. C nebo C++. Ze stejného důvodu jsou v programu často využívány konstrukce typické pro jazyk C#.

4.3 Třída HLineAccum

Jedná se o velmi jednoduchou třídu, která slouží jako akumulátor při detekci čar. Jedná se v podstatě o dvojrozměrné pole celočíselných proměnných, které nesou informaci o intenzitě daného místa. První dimenze pole představuje úhel θ, který je rozdělen do předem určených kroků. Druhá dimenze nese informaci o vzdálenosti r od počátku.

Pole *array* je kostrou této třídy. Po vytvoření instance třídy je vyplněno nulami. Na toto pole je navázaný indexer třídy tak, aby se dalo k objektu akumulátoru přistupovat přes jeho identifikátor. Tento indexer virtuálně posouvá počátek pole do středu druhé dimenze představující vzdálenost r. Je tomu tak proto, aby se dal akumulátor jednoduše indexovat jak kladnými tak i zápornými čísly, viz Obrázek 11.



Obrázek 11. Rozdíl akumulátoru uvnitř a navenek.

Modrá barva vyznačuje orientaci vnitřního pole, zatímco zelená barva ukazuje, jak se akumulátor jeví navenek při přístupu přes indexer. Dimenze úhlu θ je v obou případech totožná.

Proměnné *ThetaDimension* a *RadiusDimension* vrací rozměry akumulátoru. RadiusDimension ve skutečnosti vrací pouze poloviční hodnotu skutečné velikosti akumulátoru, protože se jedná o maximální absolutní hodnotu vzdálenosti r. Tyto proměnné nelze po vytvoření instance třídy měnit.

Proměnná *MaxIntensity* vrací intenzitu nejsilnějšího maxima v akumulátoru. Počítá se postupně při přičítání do akumulátoru. Předpokládá pouze růst hodnot v poli, jinak nemusí být správně.

Konstriktor třídy požaduje dva parametry, určující velikost akumulátoru, přičemž velikost druhé dimenze je ve skutečnosti dvojnásobná. Akumulátor je po vytvoření naplněn nulami.

4.4 Třída HLineTranform

Tato třída je centrálním prvkem detekce. Vstupními parametry je velikost úhlu, který určuje počet iterací pro jeden vstupní bod, práh intenzity vstupních dat a minimální relativní intenzita nalezených čar, které chceme dostat na výstupu. Na vstupu je samozřejmě také obrázek (bitmapa), ve kterém budeme čáry hledat. Výstupem je pak seznam přímek, které byly nalezeny během detekce a které splňují podmínku minimální relativní intenzity.

Proměnná *ThetaStep* je klíčovým parametrem celé detekce. Určuje, kolik iterací se provede během zpracování jednoho vstupního bodu. Tento úhel se zadává v radiánech a musí být v rozpětí 0 až π. Úhel může být jakékoli desetinné číslo, ovšem z důvodů optimálního prozkoumání celého obrázku se tento úhel po přiřazení změní tak, aby mohl být dělitelem čísla π. Záporný úhel se bere jako by byl kladný. Při přiřazení do této proměnné se rovněž vypočítá velikost první dimenze vnitřního akumulátoru. Změnou tohoto úhlu se znehodnocují případná data již proběhlé detekce. Při vytvoření instance třídy je tato proměnná nastavena na výchozí hodnotu $\frac{\pi}{128}$.

Proměnná *Treshold* určuje práh intenzity ve vstupních datech. Algoritmus detekce bude ignorovat všechny vstupní body, které budou pod tímto prahem intenzity. Hodnoty se mohou pohybovat mezi 0 a 255, přičemž ale detekce předpokládá vstupní formát 256 stupňů šedé. Výchozí hodnota prahu je 128. Změnou tohoto parametru se znehodnocují případná data již proběhlé detekce.

Standardní konstruktor vytvoří instanci třídy a nastaví proměnné ThetaStep a Treshold na výchozí hodnoty. Je možné také využít přetížený konstruktor, který umožňuje přímo nastavení těchto hodnot již při vytváření instance.

Metoda *ProcessImage* je první kterou je třeba zavolat po nastavení parametrů detekce. Tato metoda má jako vstupní parametr bitmapu se vstupními daty. Předpokládá se, že bitmapa má formát 256 odstínů šedi. Pokud tomu tak nebude, prahování nebude fungovat správně. Prvním krokem této metody je spočtení úhlopříčky obrázku. Tato hodnota určuje druhou dimenzi interního akumulátoru. Následně se vytvoří akumulátor. Následuje cyklus detekce, který bude plnit akumulátor. Pro každý bod ve vstupním obrázku, který bude nad prahem intenzity, se provede cyklus, který představuje průchod obrázkem v úhlech 0 až π. Tento cyklus se bude opakovat přesně tolikrát, jaký je rozměr první dimenze akumulátoru. Tento rozměr je původně vypočten z proměnné ThetaStep. V tomto cyklu se vždy nejprve vypočte skutečný úhel θ vynásobením proměnné ThetaStep iterační proměnnou cyklu. Následně se vypočítá vzdálenost r přímky od počátku dosazením do rovnice $r = x \cdot cos\theta + y \cdot sin\theta$. V tomto vnitřním cyklu se v podstatě kolem vstupního bodu „nakreslí“ přímky a jejich parametry se uloží do akumulátoru. Tato metoda v každém svém vnějším cyklu volá metodu Application.DoEvents(), proto aby nedocházelo k „zaseknutí“ vlákna. Na konci této metody se označí akumulátor jako naplněný.

V této chvíli můžeme zavolat metodu *GetAccumAsImage*, která vrací akumulátor ve formě bitmapy. Tato metoda nejprve vytvoří prázdnou bitmapu. Poté projde celý akumulátor a pro každou hodnotu vloží na příslušné místo do bitmapy odstín šedé, odpovídající relativní intenzitě dané hodnoty v akumulátoru. To znamená, že nejvyšší hodnota v akumulátoru bude zobrazena jako čistě bílá a ostatní hodnoty budou zobrazeny relativně k této hodnotě. Tato metoda rovněž volá ve svém vnějším cyklu metodu Application.DoEvents(), aby se předešlo zaseknutí vlákna.

Metoda *GetLinesByIntensity* vrací seznam nalezených přímek. Vstupním parametrem je minimální relativní intenzita přímek, které chceme v našem výsledném seznamu. Tato hodnota se pohybuje od 0 do 1. Tato metoda projde celý akumulátor a pro každou hodnotu nejprve vypočítá relativní intenzitu. Pokud tato intenzita vyhovuje podmínce minimální relativní intenzity, vytvoří se nová instance třídy HLine, do které se vloží parametry přímky. Tato instance se pak vloží do seznamu (instance třídy HLineCollection). Na závěr se vrátí tento seznam přímek. Tato metoda rovněž ve svém vnějším cyklu volá Application.DoEvents().

4.5 Třídy uživatelského rozhraní

Třída **MainWindow** je hlavní okno aplikace. Umožňuje otevření a zobrazení obrázku ze souboru a poskytuje možnost upravit jeho zobrazení. Hlavním úkolem této třídy je příprava vstupních dat pro detekci čar a následného zobrazení výsledkového okna. Také zobrazuje návod k použití.

Po vybrání detekce čar v hlavním menu se nejprve zobrazí dialog, ve kterém se zadají vstupní parametry. Po uzavření tohoto dialogu se kód dělí na dvě větve podle toho, jaké knihovny se k detekci použijí. Při detekci pomocí výše popsaných tříd se nejprve do proměnné *duration* uloží hodnota Environment.TickCount, která určuje, kolik milisekund uběhlo od spuštění operačního systému. Následně se vstupní obrázek zpracuje detektorem hran, který je součástí knihoven AForge.NET. Poté se vytvoří instance třídy HLineTransform, ve které se nastaví vstupní parametry podle hodnot zadaných v dialogovém okně. Zavolá se metoda ProcessImage a předá se jí vstupní bitmapa. Po jejím zpracování se zavolají metody GetAccumAsImage a GetLinesByIntensity. Poté se opět odečte hodnota Environment.TickCount, abychom zjistili, jak dlouho detekce trvala. Posledním krokem je vytvoření výsledkového okna a vložení do něj výstupů detekce.

V případě, že pro detekci využijeme knihoven AForge.NET, budeme postupovat obdobně. Nejprve tedy provedeme detekci hran. Poté převedeme vstupní parametry do jednotek, které tyto třídy očekávají a vložíme je. Poté provedeme detekci a vyzvednutí výstupů. Měření doby detekce je totožné. Z důvodů, které byly popsány v podkapitole 4.1, nyní musíme převézt výsledné přímky do souřadného systému, který používají třídy programu Detektor tak, aby bylo možno použít statickou metodu DrawLines třídy HLine. Postup je popsán v podkapitole 4.1. Na konec opět vytvoříme, naplníme a zobrazíme výsledkové okno.

Třída **ResultWindow** je výsledkové okno. Zde se zobrazují výsledky detekce. Okno umožňuje zobrazit buď původní obrázek, vstupní bitmapu nebo akumulátor. Do původního obrázku a vstupní bitmapy vykresluje (podle nastavení v menu) nalezené přímky. Tyto výstupy se do okna vkládají pomocí metody *InsertData*, která požaduje jméno souboru obrázku, popis vstupních parametrů, původní obrázek, vstupní bitmapu, obrázek akumulátoru a nalezené čáry. Těchto výsledkových oken může být v aplikaci otevřeno libovolný počet současně. Stejně tak jak hlavní okno umožňuje i toto upravit zobrazení obrazu podle velikosti okna.

Třída **DetectDialog** je dialogové okno, kde se nastavují vstupní parametry detekce. Zde si vybíráme, zda chceme pro detekci použít výše popsané třídy nebo chceme použít knihovny AForge.NET.
Nastavuje se zde krok úhlu θ, ale pouze dělitel čísla π, což ovšem nevadí z důvodů popsaných podkapitole 4.4. Hodnoty se mohou pohybovat od $\frac{\pi}{1}$ do $\frac{\pi}{2048}$. Dále se zde nastavuje práh intenzity, jehož hodnoty se pohybují od 1 do 255. Nakonec se zde nastavuje minimální relativní intenzita přímky, kterou chceme, aby nám detektor vrátil.

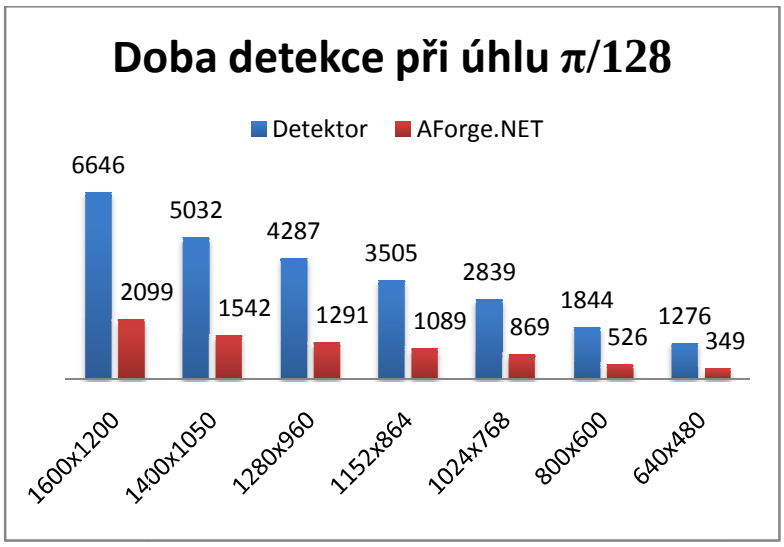
5 Porovnání výsledků detekce

Nyní se dostáváme do fáze zhodnocení implementace v jazyce C#. Nejprve budeme hodnotit průběh detekce jako celku, poté se podíváme detailněji na některé její části. Nakonec se pokusíme tuto implementaci srovnat s implementací knihoven AForge.NET. Tady je ale třeba si uvědomit zásadní věc a to sice že na knihovnách AForge.NET pracuje skupina zkušených lidí, kteří mají bohaté zkušenosti z oblasti počítačové grafiky. Proto je rozumné očekávat, že moje vlastní implementace se co do časové náročnosti nebude příliš přibližovat výsledkům AForge.NET. Další podstatná věc je to, že autoři knihoven AForge.NET jistě implementují své algoritmy za pomocí takových konstrukcí, které jsou nejméně časově náročné. Tak tomu však nebylo při implementaci mých algoritmů, ve kterých jsem se snažil spíše využít všech programátorsky přívětivých konstrukcí jazyka C#.

Další stránkou věci je kvalita a struktura výstupních dat. Toto je důležitá stránka věci, protože kvalita nalezených výstupních dat je do značné míry závislá na způsobu jejich dalšího využití. Je třeba se například zamyslet nad tím, zda dosahovaná nižší časová náročnost není na úkor kvality výsledku. Pokud např. při zpracovávání vstupních dat použijeme špatné nebo špatně nastavené druhy filtrů, může dojít sice k snížení celkové časové náročnosti detekce, ale nalezené čáry následně nemusí vyhovovat našim představám. Toto srovnání je však jen těžko vyjádřitelné číselně.

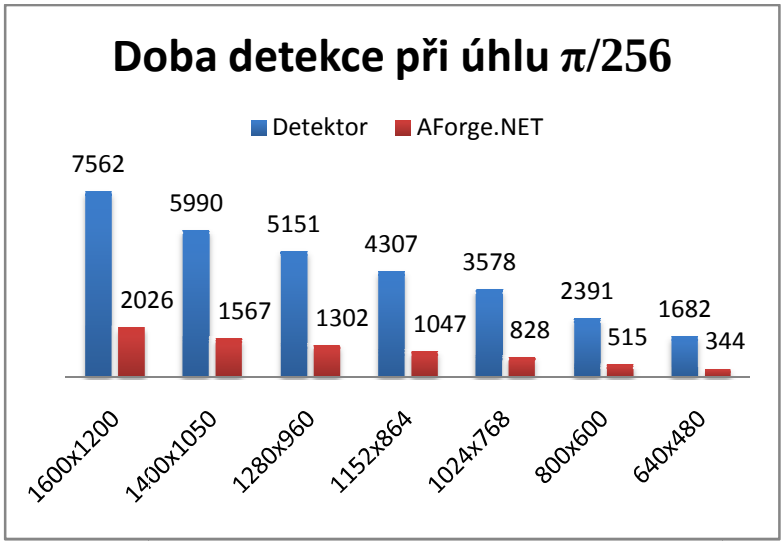
5.1 Porovnání časové náročnosti

Při porovnávání časové náročnosti celé detekce budeme postupovat následovně. Stanovíme si jednotné vstupní parametry pro testování a tři různé úhly θ. Vybereme vstupní obrázek takový, který budeme mít připravený v různých rozlišeních. Nad tímto obrázkem provedeme detekci *třikrát*, pro každou implementaci (Detektor a AForge.NET) a následně vypočteme průměr těchto tří hodnot. Tyto průměry pro obě implementace zaneseme do tabulky. Tento postup opakujeme pro všechna dostupná rozlišení, ve kterých máme připravený testovací obrázek. Graf 1 ukazuje výsledky měření pro úhel $\theta = \frac{\pi}{128}$. Délka trvání detekce je uvedena v milisekundách.



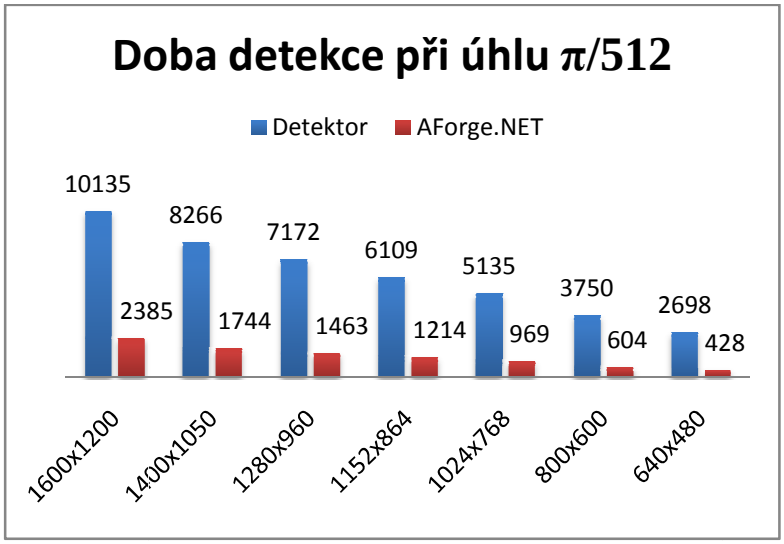
Graf 1. Doba detekce při úhlu π/128.

Můžeme vidět, že AForge.NET implementace dosahuje v celém spektru rozlišení 3x až 4x kratší doby detekce. Rozdíl mezi oběma implementacemi se zdá být téměř konstantní, až mírně klesající s vyšším rozlišením. Samotná doba detekce se zvyšuje velmi úměrně podle počtu bodů ve vstupním obrázku, ale ne tak strmě. Toto je způsobeno tím, že jakmile projde obrázek detektorem hran, počet bodů se výrazně sníží a pro větší obrázek je redukce bodů větší, než pro obrázek malý. Graf 2 ukazuje srovnání doby detekce pro úhel $\theta = \frac{\pi}{256}$.



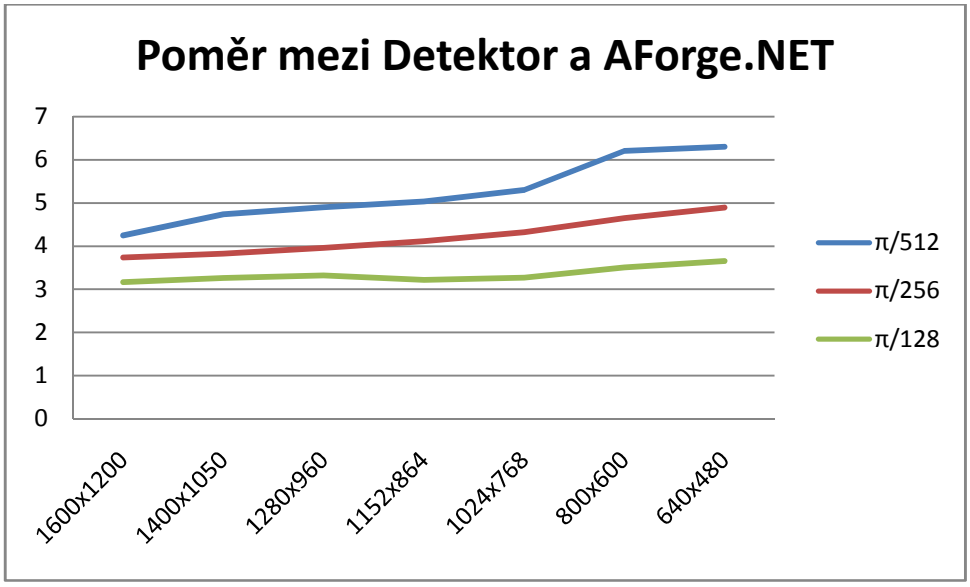
Graf 2. Doba detekcemi úhlu π/256.

Zde je situace velmi obdobná. Je zde ovšem možné pozorovat větší rozdíl mezi implementacemi než v předchozím případě. Rozdíl se navíc také více zvětšuje při nižším rozlišením. Povšimněme si, že časy pro AForge.NET jsou v podstatě identické. Toto může být důsledek vnitřní optimalizace. Naproti tomu časy pro Detektor se zvýšili v průměru o *23,3%*. Další efekt, který je možno pozorovat, je menší rozdíl mezi rozlišeními u implementace Detektor. Toto je způsobeno tím, že algoritmus věnuje více času jednotlivým bodům ve vstupních datech a tím se snižuje vliv rozlišení na výslednou dobu. Podívejme se na Graf 3, který ukazuje situaci pro úhel $\theta = \frac{\pi}{512}$.



Graf 3. Doba detekce při úhlu π/512.

Pro tento úhel je rozdíl mezi implementacemi ještě výraznější. V podstatě můžeme pozorovat všechny výše popsané jevy. Navíc však se zde vyskytuje vyšší rozdíl mezi implementacemi pro nižší rozlišení. Na nižším rozlišení je AForge.NET více než 6x rychlejší, na vyšších rozlišeních je to o něco málo více než 4x. Průměrný nárůst doby pro AForge.NET oproti předchozímu grafu je *16,6%*, zatímco nárůst pro Detektor se vyhoupl na průměrných *44,8%*. Tento fakt ukazuje, že algoritmus programu Detektor je daleko více závislý na vstupním úhlu θ. Graf 4 zachycuje kolikrát rychlejší je AForge.NET pro jednotlivá rozlišení a úhly.



Graf 4. Poměr mezi implementací Detektor a AForge.NET.

Konfigurace testovacího systému: Pentium M 740 (1.7GHz), 512 MB RAM, Intel 900 integrovaná VGA, Windows Vista Business CZ.

5.2 Porovnání paměťové náročnosti

Měření paměťové náročnosti je poněkud problematičtější. Bez velmi sofistikovaných nástrojů můžeme jen stěží srovnávat porovnatelné části kódu. Navíc je možné se domnívat, že většina paměti bude pro většinu obrázků použita spíše na režii potřebných systémových prostředků a knihoven pro uživatelské rozhraní. Měřitelný rozdíl tak bude možno sledovat až při opravdu velkých obrázcích, které ale většinou nejsou předmětem detekce čar. Můžeme však pro účel našeho experimentu takovýto scénář použít.

Budeme tedy provádět detekci hran nad obrázkem o rozlišení 1600x1200, který bude zpracován detektorem hran, nikoli však prahováním, abychom co nejvýše navýšili množství vstupních bodů. Vstupní úhel bude $\theta = \frac{\pi}{2048}$. Spustíme detekci a pomocí systémového nástroje *taskmgr.exe* budeme sledovat hodnotu využití paměti našeho procesu.

Po spuštění programu a otevření obrázku se využití paměti pohybuje někde kolem 13000 kB. Spustíme detekci. Pro AForge.NET byla nejvyšší hodnota využití paměti přibližně 24000 kB, což znamená **11000 kB** pro samotnou detekci. Pro Detektor byla nejvyšší hodnota zhruba 38000 kB, což je **25000 kB** pro detekci.

Přesto že se jedná o opravdu velmi přibližnou metodu měření, můžeme konstatovat, že algoritmus detekce čar programu Detektor je o **více než 100%** náročnější na paměť než knihovny AForge.NET. Opět se s největší pravděpodobností jedná o vnitřní optimalizaci.

6 Závěr

V úvodu textu jsem podtrhoval, že cílem této práce je zhodnotit především časovou náročnost implementace detekce čar v jazyce C#. Za tímto účelem byl napsán algoritmus využívající některé pro jiné jazyky nestandardní konstrukce, které jazyk C# umožňuje. Celkové výsledky nejsou překvapující, jde však spíše o jejich konkrétní číselné vyjádření.

Tento algoritmus je v průměru **4,2x** náročnější, než implementace knihoven AForge.NET. Tento rozdíl se zvyšuje se zmenšujícím se vstupním úhlem θ, což znamená vyšší přesnost detekce. Naopak rozdíl se zmenšuje se stoupajícím rozlišením.

Co se paměťové náročnosti týče, tak jak již bylo řečeno, přesné měření není za normálních okolností možné. Z provedených experimentů se však dá bezpečně předpokládat, že algoritmus programu Detektor je přinejmenším **2x** tak paměťově náročný než implementace knihoven AForge.NET.

Při pohledu na výsledky měření je třeba mít na paměti, že implementace AForge.NET vyšla z rukou zkušených lidí na poli počítačové grafiky a jistě obsahuje nemalé množství optimalizací. I přes tyto skutečností usuzuji, že implementovat Houghovu transformaci v jazyce C# na platformě Microsoft .NET se zřejmě *nevyplatí*. Při použití procedurálních jazyků bychom došli pravděpodobně lepším výkonům, použití objektového jazyka má však své kouzlo a lze jej v jistých situacích bez problému použít.

Tato práce se nedostala k srovnání implementací Houghovi transformace pro kružnice, elipsy a jiné pravidelné křivky. Tady je jednoznačně prostor pro budoucí vývoj projektu. Mohlo by se ukázat, že pro složitější varianty Houghovi transformace jsou rozdíly mezi procedurální a objektovou implementaci podstatně nižší.

Literatura

[1] Russ, John C.: The Image Processing Handbook. Boca Raton, Florida, USA: c1999.

[2] Wikipedia, the free encyclopedia, Hough Tranform. http://en.wikipedia.org/wiki/Hough_transform.

[3] Image Processing Learning Resources, Morphology. http://homepages.inf.ed.ac.uk/rbf/HIPR2/morops.htm.

[4] Image Processing Learning Resources, Hough Transform. http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm.

Seznam příloh